Chapter 4
SPRINTS

Scrum organizes work in iterations or cycles of up to a calendar month called sprints. This chapter provides a more detailed description of what sprints are. It then discusses several key characteristics of sprints: They are timeboxed, have a short and consistent duration, have a goal that shouldn't be altered once started, and must reach the end state specified by the team's definition of done.

Overview

Sprints are the skeleton of the Scrum framework (see Figure 4.1).

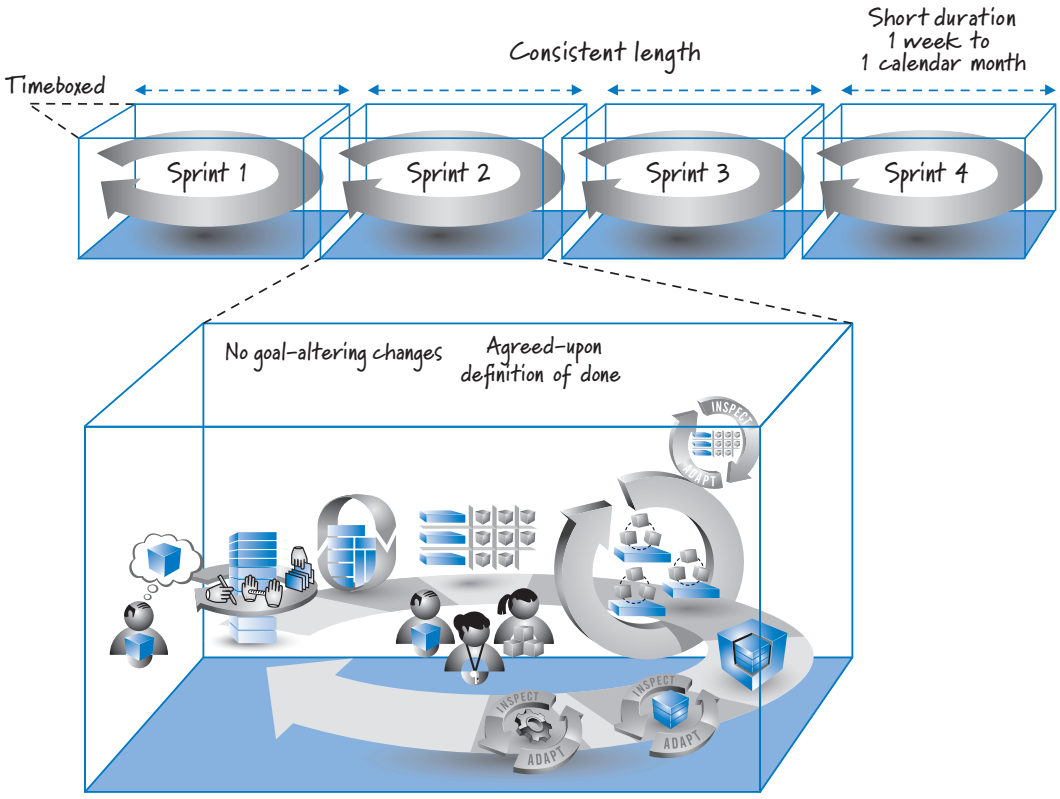

FIGURE 4.1 Sprints are the skeleton of the Scrum framework.

The gray main looping arrow in the figure, which stretches from the product backlog through the sprint execution loop and encompasses the Scrum team members, represents the sprint, on which the other Scrum artifacts and activities are shown oriented by their relative time of occurrence within the sprint. Although sprint execution is frequently confused with being "the sprint," it's really just one activity that occurs during the sprint, along with sprint planning, sprint review, and the sprint retrospective.

All sprints are timeboxed, meaning they have fixed start and end dates. Sprints must also be short, somewhere between one week and a calendar month in length. Sprints should be consistent in length, though exceptions are permitted under certain circumstances. As a rule, no goal-altering changes in scope or personnel are permitted during a sprint. Finally, during each sprint, a potentially shippable product increment is completed in conformance with the Scrum team's agreed-upon definition of done.

Although each organization will have its own unique implementation of Scrum, these sprint characteristics, with a few exceptions that we'll explore, are meant to apply to every sprint and every team. Let's look at each in detail so that we can understand why this is so.

Timeboxed

Sprints are rooted in the concept of **timeboxing**, a time-management technique that helps organize the performance of work and manage scope. Each sprint takes place in a time frame with specific start and end dates, called a timebox. Inside this timebox, the team is expected to work at a sustainable pace to complete a chosen set of work that aligns with a sprint goal.

Timeboxing is important for several reasons (see Figure 4.2).

Establishes a WIP Limit

Timeboxing is a technique for limiting the amount of WIP (work in process). WIP represents an inventory of work that is started but not yet finished. Failing to properly manage it can have serious economic consequences. Because the team will plan to work on only those items that it believes it can start and finish within the sprint, timeboxing establishes a WIP limit each sprint.

Forces Prioritization

Timeboxing forces us to prioritize and perform the small amount of work that matters most. This sharpens our focus on getting something valuable done quickly.

Demonstrates Progress

Timeboxing also helps us demonstrate relevant progress by completing and validating important pieces of work by a known date (the end of the sprint). This type of

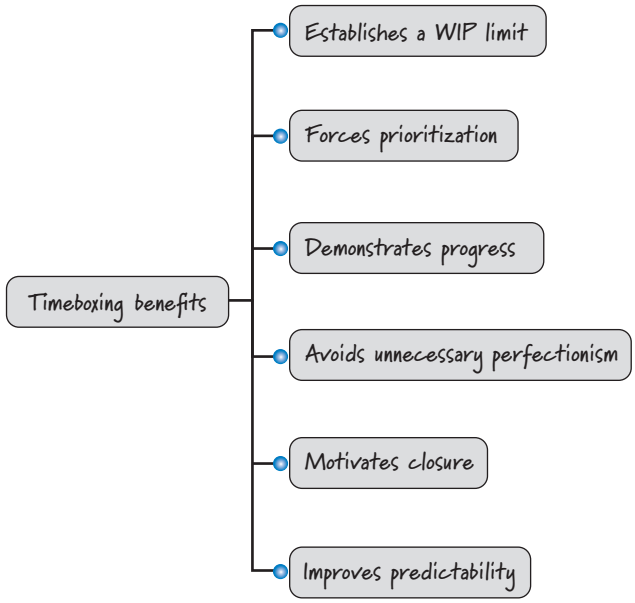

FIGURE 4.2 The benefits of timeboxing

progress reduces organizational risk by shifting the focus away from unreliable forms of progress reporting, such as conformance to plan. Timeboxing also helps us demonstrate progress against big features that require more than one timebox to complete. Completing some work toward those features ensures that valuable, measurable progress is being made each sprint. It also helps the stakeholders and team learn exactly what remains to be done to deliver the entire feature.

Avoids Unnecessary Perfectionism

Timeboxing helps avoid unnecessary perfectionism. At one time or another we have all spent too much time trying to get something "perfect" or to do "gold plating" when "good enough" would suffice. Timeboxing forces an end to potentially unbounded work by establishing a fixed end date for the sprint by which a good solution must be done.

Motivates Closure

Timeboxing also motivates closure. My experience is that things are more likely to get done when teams have a known end date. The fact that the end of the sprint brings with it a hard deadline encourages team members to diligently apply themselves to complete the work on time. Without a known end date, there is less of a sense of urgency to complete the job.

Improves Predictability

Timeboxing improves predictability. Although we can't predict with great certainty exactly the work we will complete a year from now, it is completely reasonable to expect that we can predict the work we can complete in the next short sprint.

Short Duration

Short-duration sprints provide many benefits (see Figure 4.3).

Ease of Planning

Short-duration sprints make it easier to plan. It is easier to plan a few weeks' worth of work than six months' worth of work. Also, planning on such short time horizons requires far less effort and is far more accurate than longer-horizon planning.

Fast Feedback

Short-duration sprints generate fast feedback. During each sprint we create working software and then have the opportunity to inspect and adapt what we built and how we built it. This fast feedback enables us to quickly prune unfavorable product paths or development approaches before we compound a bad decision with many follow-on

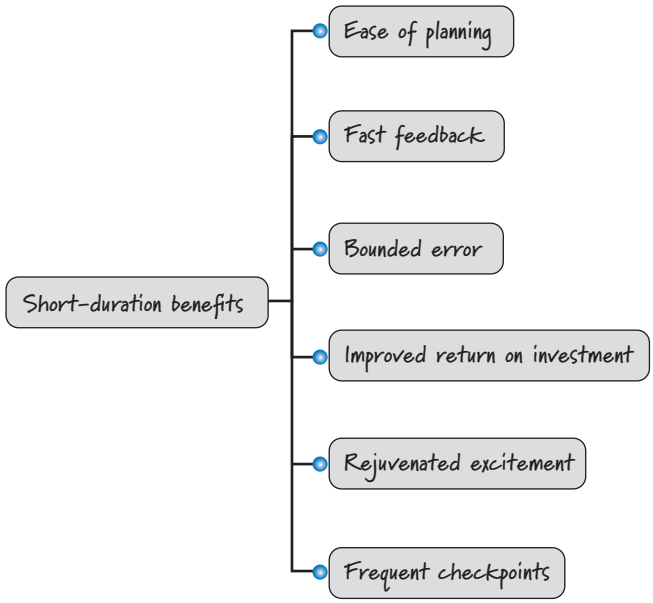

FIGURE 4.3 The benefits of short-duration sprints

decisions that are coupled to the bad decision. Fast feedback also allows us to more quickly uncover and exploit time-sensitive emergent opportunities.

Improved Return on Investment

Short-duration sprints not only improve the economics via fast feedback; they also allow for early and more frequent deliverables. As a result, we have the opportunity to generate revenue sooner, improving the overall return on investment (see Chapter 14 for an example).

Bounded Error

Short-duration sprints also bound error. How wrong can we be in a two-week sprint? Even if we fumble the whole thing, we have lost only two weeks. We insist on short-duration sprints because they provide frequent coordination and feedback. That way, if we're wrong, at least we're wrong in a small way.

Rejuvenated Excitement

Short-duration sprints can help rejuvenate excitement. It is human nature for interest and excitement to decline the longer we have to wait for gratification (see Figure 4.4).

If we work on a very long-duration project, not only are we more likely to fail; we are also more likely to eventually lose enthusiasm for the effort. (When I worked at IBM, we used to call these the "boil-the-ocean" projects, because they would

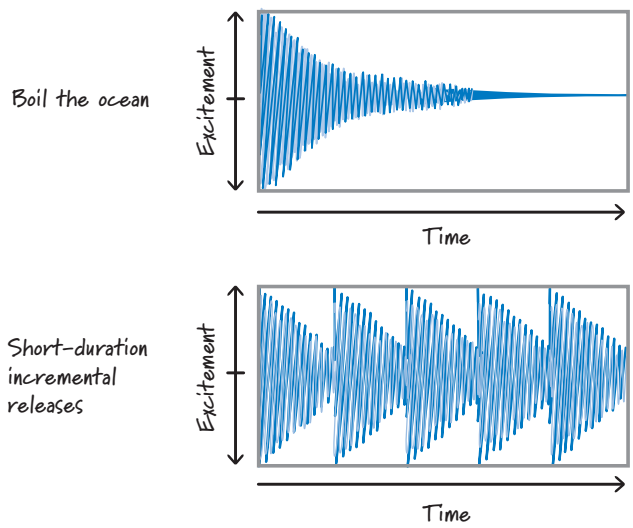

FIGURE 4.4 Excitement over time

take a really long time and a lot of effort to complete, if ever, like trying to boil an ocean.) With no visible progress and no end in sight, people begin to grow disinterested. Toward the end, they may be willing to *pay* someone to get moved to a different product!

Short-duration sprints keep participant excitement high by delivering working assets frequently. The gratification from early and frequent deliverables rejuvenates our interest and our desire to continue working toward the goal.

Frequent Checkpoints

Short-duration sprints also provide multiple, meaningful checkpoints (see Figure 4.5).

One valued aspect of sequential projects is a well-defined set of milestones. These milestones provide managers with known project-lifecycle checkpoints that are usually tied to go/no-go funding decisions for the next phase. Although potentially useful from a governance perspective, as I discussed in Chapter 3 these milestones give an unreliable indication of the true status of customer value delivery.

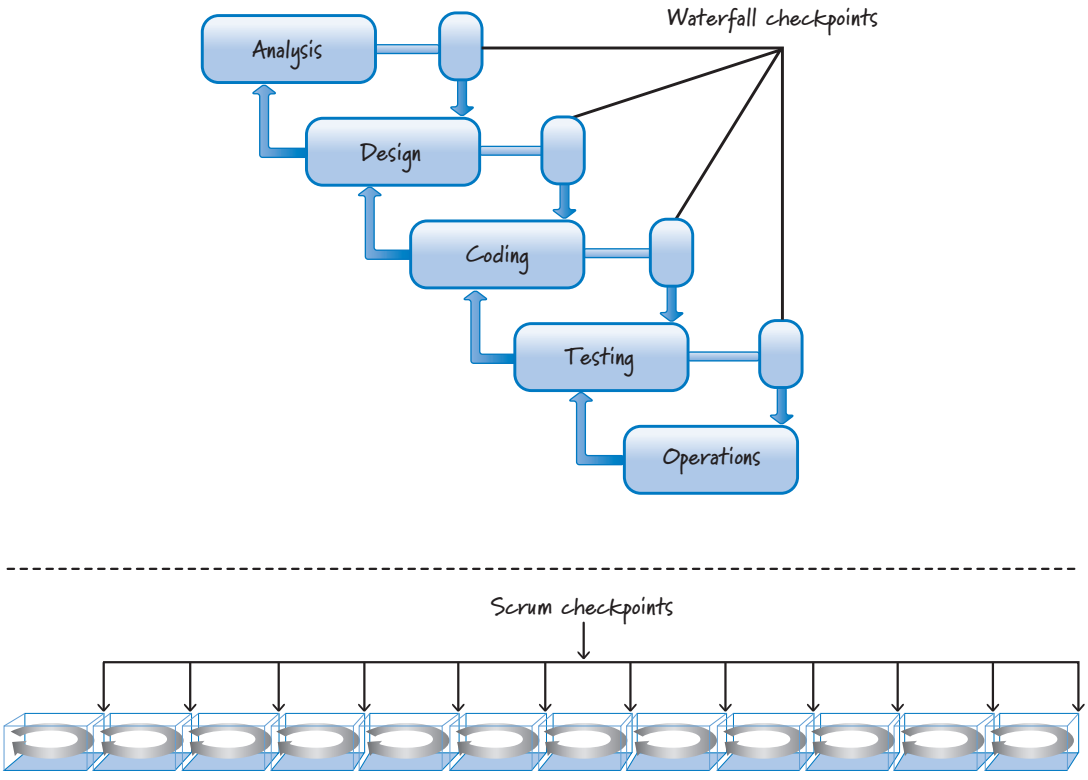

FIGURE 4.5 Checkpoint comparison

Scrum provides managers, stakeholders, product owners, and others with many more checkpoints than they would have with sequential projects. At the end of each short sprint there is a meaningful checkpoint (the sprint review) that allows everyone to base decisions on demonstrable, working features. People are better able to deal with a complex environment when they have more actionable checkpoint opportunities to inspect and adapt.

Consistent Duration

As a rule, on a given development effort, a team should pick a consistent duration for its sprints and not change it unless there is a compelling reason. Compelling reasons might include the following:

- You are considering moving from four-week sprints to two-week sprints in order to obtain more frequent feedback but want to try a couple of two-week sprints before making a final decision.
- The annual holidays or end of the fiscal year make it more practical to run a three-week sprint than the usual two-week sprint.
- The product release occurs in one week, so a two-week sprint would be wasteful.

The fact that the team cannot get all the work done within the current sprint length is not a compelling reason to extend the sprint length. Neither is it permissible to get to the last day of the sprint, realize you are not going to be done, and lobby for an extra day or week. These are symptoms of dysfunction and opportunities for improvement; they are not good reasons to change the sprint length.

As a rule, therefore, if a team agrees to perform two-week sprints, all sprints should be two weeks in duration. As a practical matter, most (but not all) teams will define two weeks to mean ten calendar weekdays. If there is a one-day holiday or training event during the sprint, it reduces the team's capacity for that sprint but doesn't necessitate a sprint length change.

Using the same sprint length also leverages the benefits of cadence and simplifies planning.

Cadence Benefits

Sprints of the same duration provide us with **cadence**—a regular, predictable rhythm or heartbeat to a Scrum development effort. A steady, healthy heartbeat allows the Scrum team and the organization to acquire an important rhythmic familiarity with when things need to happen to achieve the fast, flexible flow of business value. In my experience, having a regular cadence to sprints enables people to "get into the zone," "be on a roll," or "get into a groove." I believe this happens because regular cadence

makes the mundane but necessary activities habitual, thereby freeing up mental capacity to stay focused on the fun, value-added work.

Having a short sprint cadence also tends to level out the intensity of work. Unlike a traditional sequential project where we see a steep increase in intensity in the latter phases, each sprint has an intensity profile that is similar to that of the other sprints. As I will discuss in Chapter 11, sprint cadence enables teams to work at a sustainable pace.

Sprinting on a regular cadence also significantly reduces coordination overhead. With fixed-length sprints we can predictably schedule the sprint-planning, sprint review, and sprint retrospective activities for many sprints at the same time. Because everyone knows when the activities will occur, the overhead required to schedule them for a large batch of sprints is substantially reduced.

As an example, if we do two-week sprints on a yearlong development effort, we can send out the recurring event on everyone's calendar for the next 26 sprint reviews. If we allowed sprint durations to vary from sprint to sprint, imagine the extra effort we would need to coordinate the schedules of the stakeholders on what might be just one or two weeks' notice for an upcoming sprint review! That assumes that we could even find a time that worked for the core set of stakeholders, whose schedules are likely filled up many weeks ahead.

Finally, if we have multiple teams on the same project, having all teams with a similar sprint cadence allows for synchronization of the work across all of the teams (see Chapter 12 for a more detailed discussion).

Simplifies Planning

Using a consistent duration also simplifies planning activities. When all sprints are the same length (even when they might have a day or less capacity per sprint because of a holiday), the team gets comfortable with the amount of work that it can accomplish in a typical sprint (referred to as its **velocity**). Velocity is typically normalized to a sprint. If the length of the sprint can vary, we really don't have a normalized sprint unit. It wouldn't be meaningful to say things like "The team has an average velocity of 20 points per sprint."

While it is certainly possible to compute a team's velocity even if it uses variable-length sprints, it is more complicated. Sticking with a consistent sprint duration simplifies the computations we perform on a team's historical velocity data.

Consistent sprint durations also simplify the rest of the planning math. For example, if we are working on a **fixed-date release**, and we have consistent-duration sprints, calculating the number of sprints in the release is simply an exercise in calendar math (we know today's date, we know the release date, and we know that all sprints are the same length). If the sprint durations were allowed to vary, calculating the number of sprints in the release could be significantly more challenging (because we would have to do extensive early planning), involve unnecessary overhead, and likely be far less reliable than with consistent sprint durations.

No Goal-Altering Changes

An important Scrum rule states that once the sprint goal has been established and sprint execution has begun, no change is permitted that can materially affect the sprint goal.

What Is a Sprint Goal?

Each sprint can be summarized by a sprint goal that describes the business purpose and value of the sprint. Typically the sprint goal has a clear, single focus, such as

- Support initial report generation.
- Load and curate North America map data.
- Demonstrate the ability to send a text message through an integrated software, firmware, and hardware stack.

There are times when a sprint goal might be multifaceted, for example, "Get basic printing working and support search by date."

During sprint planning, the development team should help refine and agree to the sprint goal and use it to determine the product backlog items that it can complete by the end of the sprint (see Chapter 19 for more details). These product backlog items serve to further elaborate the sprint goal.

Mutual Commitment

The sprint goal is the foundation of a mutual commitment made by the team and the product owner. The team commits to meeting the goal by the end of the sprint, and the product owner commits to not altering the goal during the sprint.

This mutual commitment demonstrates the importance of sprints in balancing the needs of the business to be adaptive to change, while allowing the team to concentrate and efficiently apply its talent to create value during a short, fixed duration. By defining and adhering to a sprint goal, the Scrum team is able to stay focused (in the zone) on a well-defined, valuable target.

Change versus Clarification

Although the sprint goal should not be materially *changed*, it is permissible to *clarify* the goal. Let me differentiate the two.

What constitutes a change? A change is any alteration in work or resources that has the potential to generate economically meaningful waste, harmfully disrupt the flow of work, or substantially increase the scope of work within a sprint. Adding or removing a product backlog item from a sprint or significantly altering the scope of a product backlog item that is already in the sprint typically constitutes change. The following example illustrates a change:

> Product owner: "Oh, when I said that we need to be able to search the police database for a juvenile offender, I didn't just mean by last name and first name. I also meant we should be able to search the database based on a picture of the suspect's body tattoos!"

Adding the ability to search based on a picture likely represents substantially more effort and almost certainly would affect the team's ability to meet a commitment to deliver search based on last name and first name. In this case, the product owner should consider creating a new product backlog item that captures the search-by-picture feature and adding it to the product backlog to be worked on in a subsequent sprint.

What constitutes a clarification? Clarifications are additional details provided during the sprint that assist the team in achieving its sprint goal. As I will discuss in Chapter 5, all of the details associated with product backlog items might not be fully known or specified at the start of the sprint. Therefore, it is completely reasonable for the team to ask clarifying questions during a sprint and for the product owner to answer those questions. The following example illustrates a clarification:

> Development team: "When you said the matches for a juvenile offender search should be displayed in a list, did you have a preference for how that list is to be ordered?"
>
> Product owner: "Yes, sort them alphabetically by last name."
>
> Development team: "OK, we can do that."

In this manner, the product owner can and should provide clarification during the sprint.

Consequences of Change

It may appear that the no-goal-altering-change rule is in direct conflict with the core Scrum principle that we should embrace change. We do embrace change, but we want to embrace it in a balanced, economically sensible way.

The economic consequences of a change increase as our level of investment in the changed work increases (see Figure 4.6).

We invest in product backlog items to get them ready to be worked on in a sprint. However, once a sprint starts, our investment in those product backlog items has increased (because we spent time during sprint planning to discuss and plan them at a task level). If we want to make a change after sprint planning has occurred, we not only jeopardize the planning investment, but we also incur additional costs for having to replan any changes during the sprint.

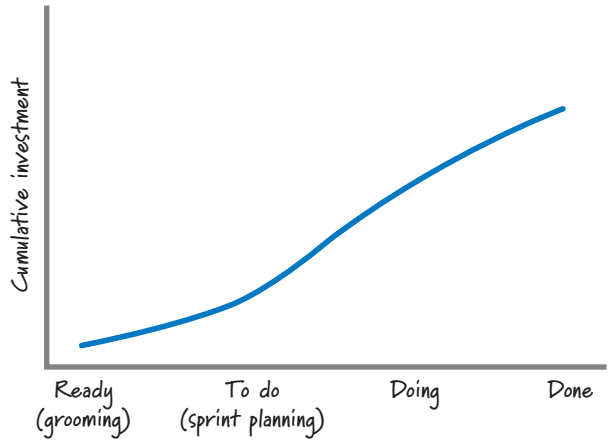

FIGURE 4.6 Cumulative investment at different states

In addition, once we begin sprint execution, our investment in work increases even more as product backlog items transition through the states of to do (work not yet started), doing (work in process), and done (work completed).

Let's say we want to swap out feature X, currently part of the sprint commitment, and substitute feature Y, which isn't part of the existing commitment. Even if we haven't started working on feature X, we still incur planning waste. In addition, feature X might also have dependencies with other features in the sprint, so a change that affects feature X could affect one or more other features, thus amplifying the effect on the sprint goal.

If work on feature X has already begun, in addition to the already-mentioned waste, we could have other potential wastes. For example, all of the work already performed on feature X might have to be thrown away. And we might have the additional waste of removing the partially completed work on feature X, which we may never use in the future (we're not going to include partially completed work in a potentially shippable product increment at the end of the sprint).

And, of course, if feature X is already completed, we might have wasted the full investment we made in feature X. All of this waste adds up!

In addition to the direct economic consequences of waste, the economics can be indirectly affected by the potential deterioration of team motivation and trust that can accompany a change. When the product owner makes a commitment to not alter the goal and then violates the commitment, the team naturally will be demotivated, which will almost certainly affect its desire to work diligently to complete other product backlog items. In addition, violating the commitment can harm the trust within the Scrum team, because the development team will not trust that the product owner is willing to stick to his commitments.

Being Pragmatic

The no-goal-altering-change rule is just that—a rule, not a law. The Scrum team has to be pragmatic.

What if business conditions change in such a way that making a change to the sprint goal seems warranted? Say a competitor launches its new product during our sprint. After reviewing the new product, we conclude that we need to alter the goal we established for our current sprint because what we are doing is now economically far less valuable given what our competitor has done. Should we blindly follow the rule of no goal-altering changes and not alter our sprint? Probably not.

What if a critical production system has failed miserably and some or all of the people on our team are the only ones who can fix it? Should we not interrupt the current sprint to fix it? Do we tell the business that we will fix the production failure first thing next sprint? Probably not.

In the end, being pragmatic trumps the no-goal-altering-change rule. We must act in an economically sensible way. Everyone on the Scrum team can appreciate that. If we change the current sprint, we will experience the negative economic consequences I previously discussed. However, if the economic consequences of the change are far less than the economic consequences of deferring the change, making the change is the smart business decision. If the economics of changing versus not changing are immaterial, no change to the sprint goal should be made.

As for team motivation and trust, in my experience, when a product owner has a frank, economically focused discussion with the team about the necessity of the change, most teams understand and appreciate the need, so the integrity of motivation and trust is upheld.

Abnormal Termination

Should the sprint goal become completely invalid, the Scrum team may decide that continuing with the current sprint makes no sense and advise the product owner to abnormally terminate the sprint. When a sprint is abnormally terminated, the current sprint comes to an abrupt end and the Scrum team gathers to perform a sprint retrospective. The team then meets with the product owner to plan the next sprint, with a different goal and a different set of product backlog items.

Sprint termination is used when an economically significant event has occurred, such as a competitor's actions that completely invalidate the sprint or product funding being materially changed.

Although the product owner reserves the option to cancel each and every sprint, in my experience it is rare that product owners invoke this option. Often there are less drastic measures that a Scrum team can take to adjust to the situation at hand. Remember, sprints are short, and, on average, the team will be about halfway through a sprint when a change-causing situation arises. Because there may be only a week or so of time left in the sprint when the change occurs, the economics of terminating may be less favorable than just staying the course. And many times it is possible to

make a less dramatic change, such as dropping a feature to allow time to fix a critical production failure, instead of terminating the sprint.

It is important to realize that terminating the sprint early, in addition to having a negative effect on morale, is a serious disruption of the fast, flexible flow of features and negates many of the benefits of consistent-duration sprints I mentioned earlier. Terminating a sprint should be the last resort.

If a sprint is terminated, the Scrum team will have to determine the length of the next sprint (see Figure 4.7).

There are three apparent options:

1. Stay with the original sprint length. This has the advantage of keeping a uniform sprint length throughout development (except for the terminated sprint, of course). If multiple Scrum teams are collaborating on the same development effort, using the original sprint length will put the Scrum team that terminated its sprint out of sync with the other teams.
2. Make the next sprint just long enough to get to the end date of the terminated sprint. For example, if the Scrum team terminated a two-week sprint at the end of the first week, the next sprint would be one week to get the team resynchronized to its original sprint cadence.
3. Make the next sprint bigger than a normal sprint to cover the remaining time in the terminated sprint plus the time for the next full sprint. So, in the previous example, make the next sprint three weeks in order to get the team resynchronized to its original sprint cadence.

In a multiteam development effort, either option 2 or option 3 would be preferred. In all cases, you will need to consider your specific context to know which option is best.

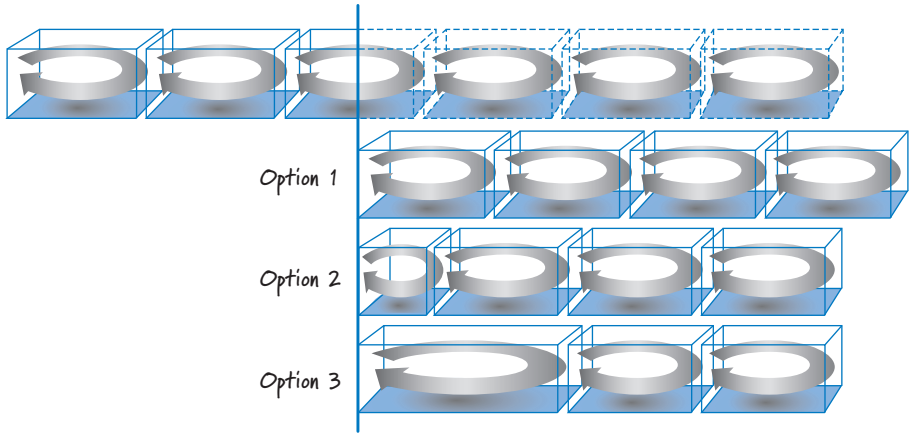

FIGURE 4.7 Deciding on the next sprint length after sprint termination

Definition of Done

In Chapter 2, I discussed how the result of each sprint should be a potentially shippable product increment. I also mentioned that "potentially shippable" doesn't mean that what was built must actually be shipped. Shipping is a business decision that often occurs at a different cadence; in some organizations it may not make sense to ship at the end of every sprint.

Potentially shippable is better thought of as a state of confidence that what got built in the sprint is actually done, meaning that there isn't materially important undone work (such as important testing or integration and so on) that needs to be completed before we could ship the results from the sprint, if shipping was our business desire. To determine if what got produced is potentially shippable, the Scrum team must have a well-defined, agreed-upon definition of done.

What Is the Definition of Done?

Conceptually the **definition of done** is a checklist of the types of work that the team is expected to successfully complete before it can declare its work to be potentially shippable (see Table 4.1).

TABLE 4.1 Example Definition-of-Done Checklist

	Definition of Done
❑	Design reviewed
❑	Code completed
❑	Code refactored
❑	Code in standard format
❑	Code is commented
❑	Code checked in
❑	Code inspected
❑	End-user documentation updated
❑	Tested
❑	Unit tested
❑	Integration tested
❑	Regression tested
❑	Platform tested
❑	Language tested
❑	Zero known defects
❑	Acceptance tested
❑	Live on production servers

Obviously the specific items on the checklist will depend on a number of variables:

- The nature of the product being built
- The technologies being used to build it
- The organization that is building it
- The current impediments that affect what is possible

Most of the time, a bare-minimum definition of done should yield a complete slice of product functionality, one that has been designed, built, integrated, tested, and documented and would deliver validated customer value. To have a useful checklist, however, these larger-level work items need to be further refined. For example, what does tested mean? Unit tested? Integration tested? System tested? Platform tested? Internationalization tested? You can probably think of many other forms of testing that are specific to your product. Are all of those types of testing included in the definition of done?

Keep in mind that if you don't do an important type of testing every sprint (say, performance testing), you'll have to do it sometime. Are you going to have some specialized sprint in the future where the only thing you do is performance testing? If so, and performance testing is essential to being "done," you really don't have a potentially shippable product increment each sprint. And even worse, when you actually do the performance testing at a later time and it doesn't go quite as planned, not only will you discover a critical problem very late in the process, but you will also have to spend much more time and money to fix it at that time than if you had done the performance testing earlier.

Sometimes the testing might take longer than the duration of a sprint. If this occurs because the development team has accrued a huge manual testing debt, the team needs to start automating its tests so that the testing can be completed within a sprint. If this occurs because of the nature of the test, we will need to accept starting the test in one sprint and finishing it in some future sprint. For example, one organization I coached was building a device composed of hardware, firmware, and software. One of its standard tests was the 1,500-hour burn-in test, where the device was run flat-out for that amount of time to see if it would fail. That test can't be completed in a two-week sprint, so the Scrum team adjusted the definition of done so that a sprint could be deemed done even if the 1,500-hour test was not yet completed.

Often I am asked, "What if there is a significant defect that remains on the last day of the sprint; is the product backlog item done?" No, it's not done! And because, as a rule, we don't extend sprints beyond the end of the planned timebox, we wouldn't extend the sprint by a day or two to fix the defect in the current sprint. Instead, at the planned end of the sprint, the incomplete product backlog item is taken from the current sprint and reinserted into the product backlog in the proper order based on the other items that are currently in the product backlog. The incomplete item might then be finished in some future sprint.

Scrum teams need to have a robust definition of done, one that provides a high level of confidence that what they build is of high quality and can be shipped. Anything less robs the organization of the business opportunity of shipping at its discretion and can lead to the accrual of technical debt (as I will discuss in Chapter 8).

Definition of Done Can Evolve Over Time

You can think of the definition of done as defining the state of the work at the end of the sprint. For many high-performance teams, the target end state of the work enables it to be potentially shippable—and that end state remains relatively constant over the development lifecycle.

For example, when I was product owner for the Scrum Alliance website redesign project in 2007, we performed one-week sprints. The end state of our definition of done could be summarized as "live on the production servers." The team and I determined that this was a perfectly reasonable state for us to achieve every sprint. We defined this end state at the beginning of the development effort; that target end state didn't change during the time I was product owner for the site.

Many teams, however, start out with a definition of done that doesn't end in a state where all features are completed to the extent that they could go live or be shipped. For some, real impediments might prevent them from reaching this state at the start of development, even though it is the ultimate goal. As a result, they might (necessarily) start with a lesser end state and let their definition of done evolve over time as organizational impediments are removed.

For example, I visited an organization that builds a clinical informatics system. Its product is installed in a medical clinic and collects a variety of clinical data (some even directly from the machines that perform diagnostic tests). The team knew that clinical testing, which involved installing the product in a clinical lab to make sure it worked with clinical hardware, would be required before they could ship. However, because they didn't have regular access to a lab, the team didn't at first include clinical testing in its definition of done. Instead, it included clinical-testing sprints at the end of each release.

In our discussions, I learned that marketing and the team hated these prerelease clinical tests. No one could predict how many sprints it would take to work out all of the defects, and the product couldn't be released until the defects were removed. As we were brainstorming potential solutions, the VP of Engineering chimed in. He asked his team, "If you had access to a clinical lab, would you be able to do clinical testing each sprint?"

The team members discussed his question and responded, "Yes, but it will mean we complete fewer features each sprint." The VP agreed to remove the impediment by getting the team access to a local university clinical lab. The product owner agreed that having fewer features completed each sprint was a sensible trade-off for knowing that the features that were delivered had been clinically tested. At that point the team

was able to evolve its definition of done to actually achieve "potentially shippable," giving everyone involved a higher degree of confidence in the work completed each sprint.

Other times a team might have an impediment that it knows can't be removed right away. As a result, it knows that the definition of done during its product development effort will necessarily evolve. A common example is a product that includes hardware and software. I have seen Scrum applied to the development of many such products, and frequently I'll hear the software people say, "The hardware always arrives late!" In cases like this, if the team is building software and it doesn't have the actual hardware on which to test the software, it can't really claim that the results produced at the end of the sprint are potentially shippable. At best it might claim "emulator done," because testing during the early sprints is typically performed against a software emulator of the actual hardware. Later, when the actual hardware is available, the definition of done will evolve to mean potentially shippable or at least something closer to it.

Definition of Done versus Acceptance Criteria

The definition of done applies to the product increment being developed during the sprint. The product increment is composed of a set of product backlog items, so each backlog item must be completed in conformance with the work specified by the definition-of-done checklist.

As I will discuss in Chapter 5, each product backlog item that is brought into the sprint should have a set of **conditions of satisfaction** (item-specific acceptance criteria), specified by the product owner. These **acceptance criteria** eventually will be verified in **acceptance tests** that the product owner will confirm to determine if the backlog item functions as desired. For example, if the product backlog item is "Allow a customer to purchase with a credit card," the conditions of satisfaction might be "Works with AmEx, Visa, and MasterCard." So each product backlog item will have its own appropriate set of acceptance criteria. These item-specific criteria are in addition to, not in lieu of, the done criteria specified by the definition-of-done checklist, which apply to all product backlog items.

A product backlog item can be considered done only when both the item-specific acceptance criteria (for example, "works with all of the credit cards") and the sprint-level definition-of-done (for example, "live on the production server") items have been met.

If it is confusing to refer to product backlog items that pass their acceptance criteria as *done*, call them *completed* or *accepted*.

Done versus Done-Done

Some teams have adopted the concept of "done" versus "done-done." Somehow done-done is supposed to be more done than done! Teams shouldn't need two different

concepts, but I have to admit to using both terms with my son and his homework. I used to ask my son if he was "done" with his homework and he would tell me yes. Then I went to parent-teacher night at his school, and during a discussion with his teacher I asked, "So, when he turns in his homework, is it done?" She said, "Not really!"

After a more probing discussion with my son, I came to understand that his definition of done was "I did as much work as I was prepared to do!" So, from that point forward I started using the term *done-done*, which we both agreed would mean "done to the point where your teacher would think you are done."

Teams that are unaccustomed to really getting things done early and often are more likely to use done-done as a crutch. For them, using done-done makes the point that being done (doing as much work as they are prepared to do) is a different state from done-done (doing the work required for customers to believe it is done). Teams that have internalized that you can be done only if you did all the work necessary to satisfy customers don't need to have two states; to them, done means done-done!

Closing

In this chapter I emphasized the crucial role of sprints in the Scrum framework. Sprints provide the essential Scrum skeleton on which most other activities and artifacts can be placed. Sprints are short, timeboxed, and consistent in duration. They are typically defined by a sprint goal, a goal that should not be altered without good economic cause. Sprints should produce a potentially shippable product increment that is completed in conformance with an agreed-upon definition of done. In the next chapter I will focus on the inputs to the sprints—requirements and their common representation, user stories.